

T2Script Programming Language

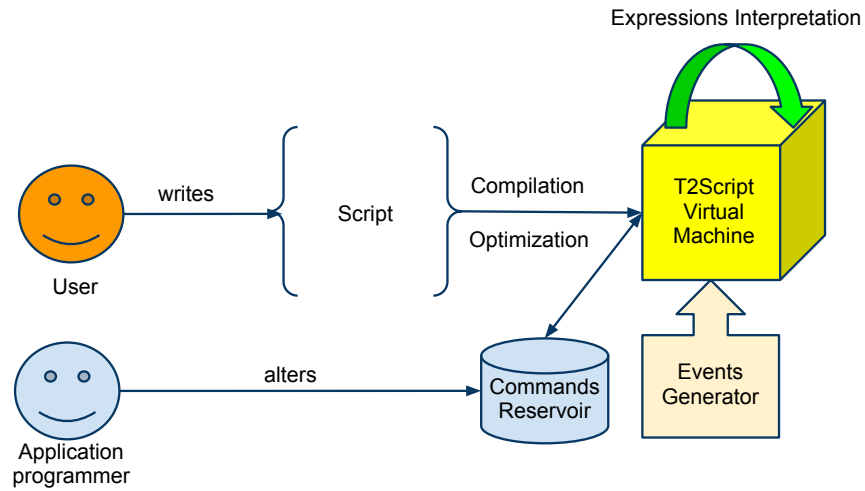
Piotr J. Puczynski [piotr@taboretsoft.com]

T2Script Research Group at TaboretSoft
 Department of Informatics and Mathematical Modeling
 Technical University of Denmark
 Richard Petersens Plads, build. 321
 DK-2800 Lyngby, Denmark

January 31, 2011

Abstract

Event-driven programming is used in many fields of modern Computer Science. In event-driven programming languages user interacts with a program by triggering the events. We propose a new approach that we denote command-event driven programming in which the user interacts with a program by means of events and commands. We describe a new programming language, T2Script, which is based on command-event driven paradigm. T2Script has been already implemented and used in one of industrial products. We describe the rationale, basic concepts and advanced programming techniques of new T2Script language. We evaluate the new language and show what advantages and limitations it has.



Contents

I	Introduction	4
1	Foreword	4
2	Motivation	4
2.1	Contribution and novelty	5
3	Taboret2 application	5
4	Language history and development	5
II	Basics	6
5	Language purpose	6
6	Scripts interpretation process	7
7	Commands	8
7.1	Format of commands	8
7.2	Case-sensitivity	9
7.3	Multi-line policy	10
7.4	Context of commands	10
7.5	Result of command	10
7.6	Expressions interpretation modes	11
7.7	Internal T2Script commands	11
7.7.1	WHILE	11
7.7.2	FOR	12
7.7.3	Eval-type commands: MECHANIZE and MLC	13
8	Expressions	14
8.1	Format of expressions	14
8.2	Variables	15
8.3	Constants	15
8.4	Complex expressions	15
8.4.1	Operators	16
9	Functions	17
9.1	Functions arguments	18
9.2	Functions result	18
9.3	Functions calls	19
10	Events	19
10.1	Events variables	20
10.2	Events results	20

11 Timers	21
III Advanced programming techniques	21
12 Exceptions handling	21
13 Regular expressions	22
14 Metaprogramming	22
15 Dynamic-Link Libraries (DLL) plugins	23
16 Debugging and Testing	23
IV Summary	23
17 Solutions to the given problems	24
17.1 Interaction with the user	24
17.2 Abstraction of an events dispatcher	24
17.3 Separation of the text and expressions	24
17.4 Other languages modules	25
17.5 Static code analysis	25
17.6 Language extensions	25
18 CED programming	25
19 Evaluation	25
20 Future work	26
References	27

Part I

Introduction

1 Foreword

This is a vision document describing fundamental concepts in a new dynamic language *T2Script*. The language, including all former and current versions, has been designed by Piotr J. Puczynski in years 2003-2010. The language is *command-event driven* and designed to allow fast purpose-specific operations for handling strings. It is also designed to be integrated with other programming languages by means of metaprogramming techniques. The language can be used for many applications in both academia and industry.

We would kindly like to thank Hubert Baumeister and Michal Staszewski for reading the first versions of the manuscript and giving a feedback. Also we would like to thank the members of community of *Taboret2* – that sometimes acted as guinea pigs for the subsequent language versions – for important feedback during the language design and for pointing out problems and possible improvements.

2 Motivation

The event-driven programming languages are usually used to support *Graphical User Interfaces* (GUI) [4], network and sometimes to implement robust or real-time systems [3, 5]. In this paper, we will show a new language, *T2Script*, which constitutes event-driven paradigm for purpose-specific language. The event-driven programming environments have many applications in telecommunications and *Human-Computer Interaction* (HID) systems. In these dynamic environments, the program needs to handle many events in a short time. Our *T2Script* language aims to solve the problems and issues:

1. In events-driven languages the interaction with the user is limited to the events. We would like to increase the scale of interaction in our language by introducing *commands* (as element of language). This approach is very useful in many of specific applications that the language could be adapted for. We will denote that approach as *command-event driven* (CED) *programming paradigm*;
2. In most of the languages like *Java*, the events dispatcher is exposed to the *user* (i.e. script programmer) as listener that is not an internal part of language itself. For simple applications, this approach may be seen as too complex for a given problem and may discourage a user to follow it.
3. There is no event-driven specific-purpose language that was designed for strings handling in which the language expressions are being separated from the text, not text being separated from the expressions (that essentially makes a difference of importance);
4. The existing languages don't allow easy extensions with modules written in other programming languages. And if they do the technique usually increases program execution time;
5. Problem of static code analysis for security, particularly, for contract enforcement;
6. Standard programming languages are difficult (if not impossible) to improve and change.

We will show our solution generically but we will also mention the example implementation of *T2Script* language that was build-in as a module to *Taboret2* application that is popular chat software in Poland (see Section 3). This application extends the commands set of *T2Script* language with *Internet Relay Chat* (IRC) client-server specific interactions.

2.1 Contribution and novelty

Few significant contributions are made by this paper. It is the first publication about the new dynamic programming language: T2Script. The document is a description revealing language elements and presenting how to use them from the point of view of the user (and, in limited scope, application programmer). It shows how the problems presented in Section 2 being solved in the language design and build-in language commands. Document also contains description of CED Programming paradigm that is proposed based on T2Script. The Section 19 contains the very first try to evaluate the new language.

Note, this is the vision document and the problems are inspected from high-level remoteness. The detailed descriptions of solutions of some problems are left for future work.

3 Taboret2 application

Taboret2¹ is chat software popular in Poland. It is designed to be the client application of *Onet.pl* extended IRC servers – the biggest Polish Internet portal and 3rd most accessed website in Poland (according to Alexa.com data from January 2011). Taboret2 was founded in 2002 and is constantly developed since.

The previous versions of T2Script language that we describe had been used in the application and had number of developers in real-world. Mostly due to program localization issues (Polish interface of Taboret2) and absence of good documentation the language is not known to the public. This document aims for a change in this subject.

4 Language history and development

The current version of T2Script was introduced for the first time in the software bundle *Taboret2 4.0* in the end of 2010. For simplicity, we will assign the language version to be *4.0*. Providentially, this assignment also corresponds to the language evolution as shown in fig. 1.

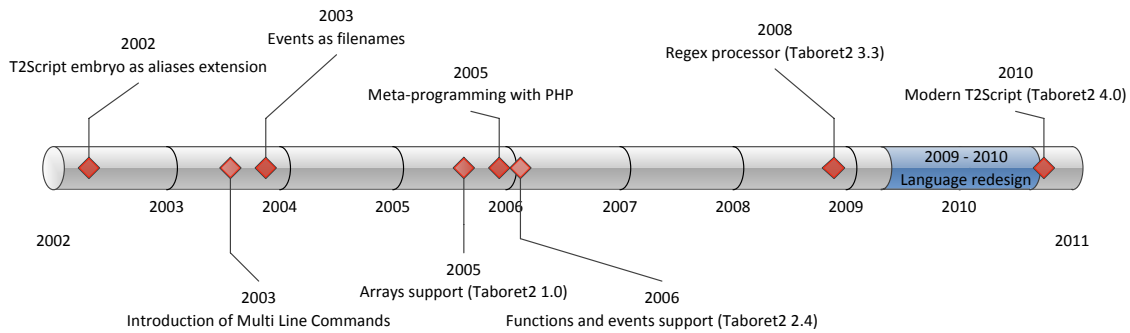


Figure 1: Timeline of Evolution of T2Script.

T2Script notation comes from “Taboret2 Script”. This name was chosen at the beginning of language development as at that time T2Script was an internal inseparable Taboret2’s part. T2Script was born in 2002 when first extension to aliases was developed in the application. Aliases were simple user customized commands redefinitions. The aliases definition required more complex commands handling in the engine.

¹Taboret2 website: <http://www.taboret2.pl>

Subsequently, users required more complex expressions for more complex aliases. All this led to introduction of *Multi Line Commands* (MLC) that encapsulated many language commands in one line (so the user would be able to input it in text field). The complexity of usage of MLCs was later solved by introduction of “scripts” saved in files. At that time (2003) also first events support (recognized on the base of filenames) was added. Functions support was added later in 2006 and solved the file names based events limitations.

Before language redesign in 2009-2010 it was not possible to reuse the language apart from Taboret2. The mentioned language redesign was a breakthrough in T2Script history. Language was now designed as a separable module of Taboret2 with possible future extension to other applications. MLCs (that were still used for some purposes before) became obsolete and the language became more expressive. Many advanced mechanisms (like exceptions handling) were added to the language.

In this document, we will describe the modern version of T2Script (T2Script 4.0) that crystallized after the 2009-2010 language redesign.

Part II

Basics

5 Language purpose

The purpose of T2Script programming language is to enable the *custom application* (later denoted also as *application*) to use commands and events interfaces. The language module is embedded and configured in custom application by *application programmer*, then used by *user* (see fig. 2).

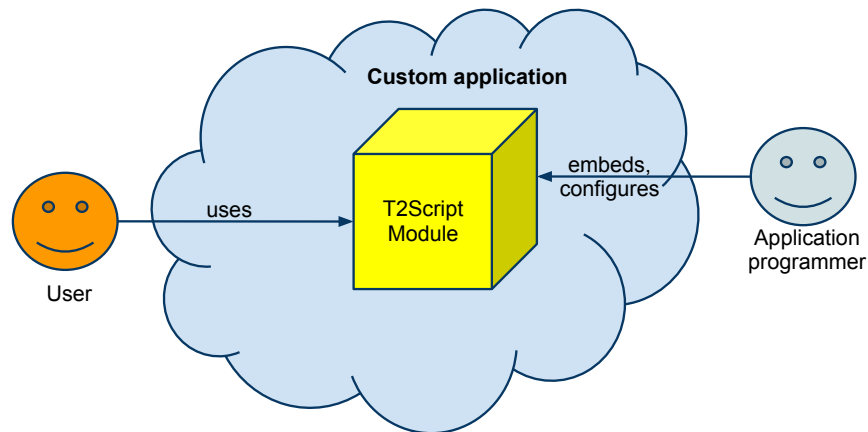


Figure 2: T2Script usage in custom application. Application programmer first embeds and configures T2Script Module in her application, then the module is used by a user.

Because commands nature is text-based, the T2Script language is mainly designed for handling strings. The idea of designing embeddable command programming language is not new. In 1989 John K. Ousterhout designed *Tool command language* (Tcl) [8]. Tcl can be reused in many software applications as suggested in [11]. In Tcl an event can be bind to application specific command. T2Script allows user to define and trigger own events, as well. Tcl is implemented in C

that may not be suitable for nowadays applications without using additional libraries. T2Script is implemented in C++ and supports object-oriented structure when embedded in application. Nevertheless, indisputably there are many similarities in the language purposes shared between Tcl and T2Script.

T2Script comprises following design objectives:

- ~> The language principal entities are *commands* and *events*. Commands and events are divided in 3 groups: language internal, application specific and user defined. The purpose of commands is to enable users of application to run them. The events are defined as actions that occur at specific circumstances in the application.
- ~> The language is programmable. Application programmer and application itself are able to change the commands set in run-time. Users and application programmers can also define new events and trigger them.
- ~> For simple usage, the language is transparent for the user. I.e. for simple commands the user may not even be aware of using the programming language structures.
- ~> The language is efficient for interpretation at run-time. It is always a trade-off between programming language expressiveness and simplicity of machine interpretation. The dilemma was mentioned in the book by John C. Mitchell that belongs to classic literature of programming languages design [7]. In case of T2Script it is important for the language expressions to be interpreted with low cost as the scripts are only precompiled and run in virtual machine. Events can also occur very often subsequently or parallelly.
- ~> The language provides a simple structured interface to be adopted in many existing programs. The time needed for embedding the language into custom application should not exceed 24 hours of work of advanced programmer.
- ~> The language allows low-cost metaprogramming techniques. This approach makes it easy to write the programs using T2Script commands in other programming languages and execute them in T2Script environment.

6 Scripts interpretation process

Scripts in T2Script are translated to internal bytecode format that is in *directly interpretable representation* (DIR) form before execution on T2Script application-level process *Virtual Machine* (VM) [12, 10]. The *VM Application Programming Interface* (API) is the current set of commands (*Commands Reservoir*) that can be altered by the application programmer (or, at run-time, by application). The general scenario of scripts running on T2Script VM is shown in fig. 3.

User supplies T2Script file in one of two forms:

1. Single-command script received from user's input (e.g. from the console);
2. Script file with functions and events definitions.

These two are different for the T2Script VM. The single command is compiled and executed immediately with specific *command's context*. The script file also contains the definitions of *functions* and *events* that cannot be supplied with single-commands. It is optimized during the translation to internal form by efficient optimization techniques [2] and command caching. Next, the translated DIR is loaded into VM (occurs only once). Then the script is set to idle state and run on VM each time *Event Generator* triggers an event. Scripts files are executed using *default context* (*context* concept is described in section 7).

The script files in T2Script currently use format .TSC and are distributed in source code. Each script file represents logically a separate module. The translation to DIR is performed after the script is loaded in the memory (e.g. at application startup or when user calls LOAD command). T2Script fully supports *Unicode* and script files can be saved using *Unicode* standard [1].

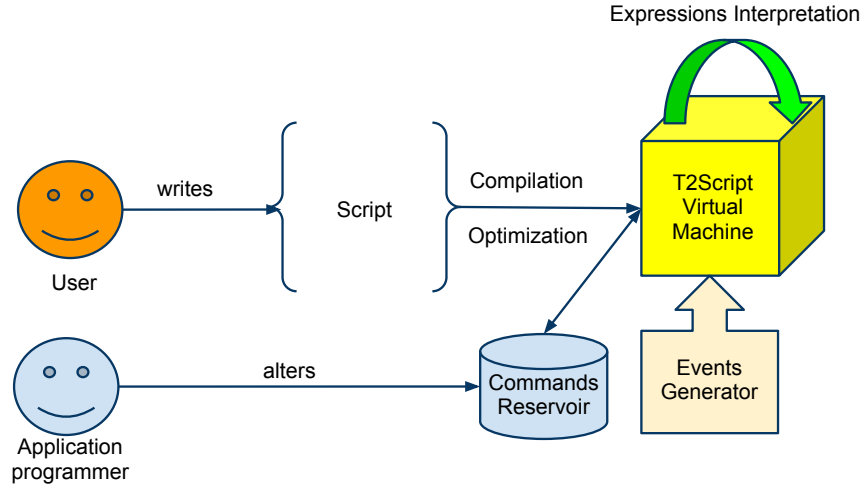


Figure 3: User and application programmer interactions with T2Script implementation elements. Note, interacts with T2Script engine by supplying script (single command or script file) and application programmer interacts by defining and altering an entries in Commands Reservoir.

During the run on T2Script VM all scripts are interpreted as they still contain simple *expressions* in the format described in section 8. This process has the nethermost low-cost because of existence of *expressions preambles* and characteristic discriminative expressions format.

7 Commands

Commands are an underlying concept of T2Script. Commands are used as the language keywords as well as to call application defined operations.

7.1 Format of commands

The command is identified by its name. Name consists of arbitrary number of Unicode characters except space, “#”, “;”, ““”, “|” and “/”. Command accepts from 0 to many *parameters*. Fig. 4 presents the structure of command in T2Script.

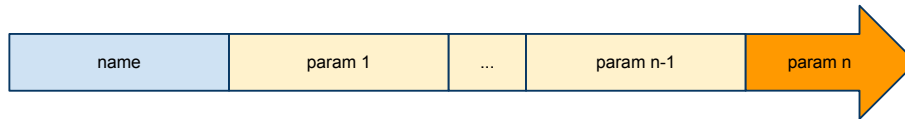


Figure 4: Structure of command accepting *n* parameters in T2Script. Identical colors of elements express similarity. The presented elements are separated with at least one space. Consequently, only the last, arrow-shaped element – parameter *n* – is allowed to contain spaces.

Every parameter and a name are separated from each other by one or more spaces. Only the last parameter of the command is allowed to contain spaces. The example of multi-parameter command is presented in listing 1. Note, the command SETVAR

accepts 2 parameters and in the example, second parameter consists of several words. The semicolon (line-ending character in file scripts) for single-command supplied from user input is skipped. In all the examples, we assume the commands are placed in the script file, and therefore they must have a semicolon after the last element.

Listing 1: Example multi-parameter command.

```
setvar my_var Hello, this is multi words parameter;
```

Second more complex and powerful command format in T2Script is block-command presented in fig. 5. Block-commands

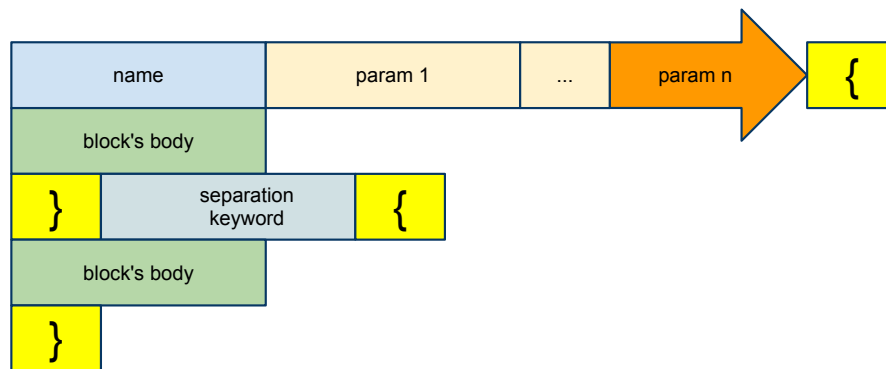


Figure 5: Structure of block-command accepting n parameters and 2 blocks. Identical colors of elements express similarity. The bodies contain other commands. The second block with separation keyword can be set as optional.

are used only in the script-files and they allow defining more sophisticated commands like language keywords. The block-command has optionally one or two blocks. The second block must have *required* or *optional* attribute. Each block contains one or more commands (including multi-parameter and block-commands). The example IF condition command (that is a block-command) in T2Script is presented in listing 2.

Listing 2: Example block-command.

```
if $success {
    setvar @text Patient fine;
} else {
    setvar @text Patient still sick;
}
```

For the presented example, ELSE is separation keyword. The keyword and the following block are optional.

7.2 Case-sensitivity

The names of the commands are case-insensitive. It is exception as all other language's elements (including variables and function names) that are case-sensitive. The rule stems from:

1. compilation process that effectively removes the names of the commands from the bytecode and replaces them with identifiers that point to VM commands functions;
2. the user input method when the language is used in *interactive mode* in the console (command name is traditionally case-insensitive in the consoles).

7.3 Multi-line policy

User can distribute command over many lines in the script file. Logical line separator is semicolon (“;”).

In normal situation, when command is distributed over many lines the space is added as a connecting character. This is avoided by adding acute (“’”) before connection (compare listing 3).

Listing 3: Command distributed over many lines with and without connector.

```
// space connector is added
setvar multi My name is
    James;

// space connector is avoided
setvar Pi 3.1415926535 ‘
    8979323846;
```

In the example, the value of *multi* is “My name is James”. Value of *Pi* is “3.14159265358979323846”.

In the blocks, the semicolon preceding the closing element of the block (“}”) and following the last command in the block can be skipped but it is not recommended. Semicolon after block-command is skipped.

We also note the comments in T2Script are always preceded by “//”. Additionally for respecting the comments, “//” must be the first non-white characters in the line.

7.4 Context of commands

Every command in T2Script contains a context. Context is background information of command execution. From the point of view of application programmer, context is an object that she defines *a priori* and attaches to the command. Context is then used to decide which commands and constants are available in the environment. For instance, in Taboret2, implementation context is IRC channel object and it sets some specific commands and constants for each window in the application in the way the user that runs the same command in different windows will get different results. Some specific commands and constants are only available for some windows.

Context can be changed at run-time by application programmer for single-commands. Dynamic changing of context functionality is not part of the language but can be exposed to the user. E.g. in Taboret2 application, this functionality is exposed to the user in the command WND that changes the window of command context.

It is optional to set the context manually. If command context is not set, command executes with default context. That is the case of commands in the text scripts from the functions.

7.5 Result of command

From the point of view of application programmer, every command returns a result. The result is a Boolean value. If the value is *false* and the command error field is set then it is considered that the error occurred during the command execution.

If the value is *false* and the value of error is not set or is set to specific constants defined in T2Script VM it has other meaning for definition of more complex language structures like BREAK mechanism in loops. Table 1 presents the existing commands results and error codes combination and their effects for languages structures build into T2Script.

Result	Error code	Effect
true	-	command successfully executed
false	-	function return
false	"continue"	next iteration of innermost loop
false	"break"	termination of innermost loop
false	(other)	propagation of an error from the command

Table 1: The result of command and error code effects used for internally build-in commands. Note, user can expand the language error codes as she wishes adding new functionality to the language. The special error codes (that are not to be displayed for the user or caught) should start with character “~”.

7.6 Expressions interpretation modes

In T2Script, each command definition must include information about the mode of the expressions interpretation. This mode affects the VM policy of handling expressions passed to the command with parameters and is either:

automatic in which VM handles all expressions virtually before passing the execution flow to the command code;

on demand in which application programmer is responsible for manual handling² of the expressions in desired data portions from parameters. In this mode, expressions are passed to the command in *raw* format (not interpreted).

This approach enables application programmer of constructing eval-type commands (like MECHANIZE that is “eval command” in T2Script).

7.7 Internal T2Script commands

T2Script defines some basic commands that are used as a language control, variables, functions and events handling, manipulation of arrays and timers, interpretation of expressions strings and numbers manipulation, loading scripts modules, invoking other processes and metaprogramming. The most significant commands that are available initially in T2Script are listed in table 2.

The application programmer or an application in which the T2Script module is embedded is able to disable or add selected commands (i.e. language structures) during configuration stage and at run-time.

We look upon some of the internal commands as constructs not present in other programming languages and therefore not intuitive for the user at a very beginning. The constructs follow strictly the format of commands presented in section 7.1.

7.7.1 WHILE

In T2Script, WHILE command is a standard *while loop* with exception it accepts 2 blocks separated with ELSE from which second block executes once if and only if the loop condition is *false* at the moment when the program flow enters the condition. The first block is standard loop’s body that executes each time the condition is *true*. Listing 4 presents the example of WHILE command usage. The TEXTOUT command is used in the example for outputting the parameter to standard output. The loop never executes NULL command that is used here as a placeholder because of the *false* control condition.

²Manually means, in this context, that programmer uses functions and objects of *VariableContext* type in the VM for handling expressions.

Command	Parameters	Blocks	Command	Parameters	Blocks
IF	1	2	FUNCTION	(no limit)	-
REPEAT	1	1	PUT	3	-
WHILE	1	2	FUNCTIONDEL	1	-
FOREACH	3	1	RETURN	1	-
FOR	3	2	RESULT	1	-
BREAK	-	-	ARGS	(no limit)	-
CONTINUE	-	-	TRIGGER	1	-
THROW	1	-	NULL	-	-
CATCH	1	1	SETVAR	2	-
MLC	1	-	DELVAR	(no limit)	-
MLCEXT	2	-	ISSET	2	-
MECHANIZE	1	-	ISNUMERIC	2	-
LOAD	2	-	SETARRAY	3	-
RUNSCRIPT	2	-	DELARRAY	1	-
ENVRS	3	-	ARRAYSIZE	2	-
EXPR	1	-	ISARRAY	2	-
SETTIMER	3	1	RUNFILE	1	-

Table 2: The most significant internally predefined T2Script language commands.

Listing 4: WHILE loop.

```
while $_false {
    null;
    // this code is never executed
} else {
    textout This code is executed only once;
}
```

7.7.2 FOR

FOR command is T2Script version of *for loop* from C++ language. FOR accepts 2 blocks separated by EVERY and 3 parameters, being the most complex internal T2Script command. First block is standard loop's body and second block is executed always after each entered loop's iteration (even if loop flow was altered with CONTINUE or BREAK commands).

The parameters to the command are:

1. a name of the variable;
2. an initial value set to variable;
3. a control condition of the loop.

The example loop is shown in listing 5.

Before loop starts, the variable i is set to 0. Then, the loop iterates 10 times until i value reaches 10. The value of i is displayed on every iteration except when the value of i is equal to 5. After each loop iteration (including the one when i

Listing 5: FOR loop.

```
for i 0 $?[< $i 10] {  
    if $?[eq $i 5] {  
        continue;  
    }  
    textout $i;  
} every {  
    inc i;  
}
```

is equal to 5), INC command increments the value of the variable by 1. After the control condition becomes *false* and loop finishes, variable *i* still exists with value 10. The presented example uses *expressions* that are described in section 8.

7.7.3 Eval-type commands: MECHANIZE and MLC

Basic eval-type command in T2Script is MECHANIZE. The command evaluates the other command provided in the parameter. Listing 6 shows the example of usage of MECHANIZE.

Listing 6: MECHANIZE command example.

```
setvar prog setvar name;  
mechanize $prog. John;
```

In the result of this program, variable *name* is set to value “John”.

The more complex version of MECHANIZE is MLC (*multi-line command*) that evaluates multiple commands provided in the parameter and separated by separator (by default “||”). User chooses MLC command when she inputs only one-line script program. Listing 7 shows the example of usage of MLC.

Listing 7: MLC command example.

```
setvar i -10;  
while $?[< $i 0]. mlc textout $i||inc i;
```

The provided example uses MLC to evaluate 2 (two) commands TEXTOUT and INC. Note, WHILE is an in-line version of WHILE that doesn’t accept blocks and executes only one instruction provided with second parameter. T2Script provides also in-line versions of commands: IF, REPEAT, WHILE and FOREACH. These versions are designed to be used with MLC commands.

8 Expressions

Expressions are used to evaluate meaningful language terms. The expressions in T2Script fall into four categories: *variables* (see section 8.2), *constants* (see section 8.3), *complex expressions* (see section 8.4) and *function calls* (see section 9.3).

8.1 Format of expressions

User injects expressions into commands parameters to update them with desired results of expressions. In the language, the user-data (including strings and numbers) is not delimited with any characters and therefore is considered of greater importance than expressions³. The concept of great importance of user-data derives from the language purpose (string processing and command processing). Consequently, the expressions must be delimited in order to separate them from the user-data.

The structure of the expression in T2Script is presented in fig. 6.

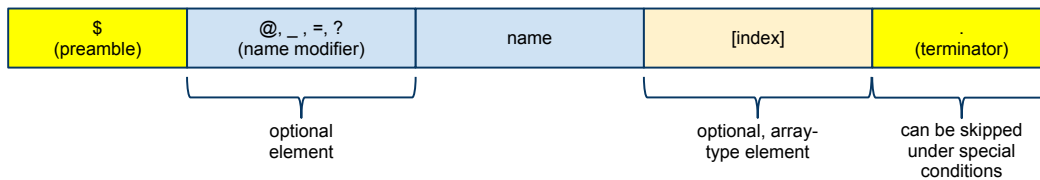


Figure 6: Structure of generic expression in T2Script. Identical colors of elements express similarity. The expression is delimited with preamble (dollar sign) and terminator (dot). If used, name modifier is one of the following symbols: @, _, =, ?. Array-type expressions have index between square brackets.

We can distinguish few elements of expression:

preamble (dollar sign) introduces the start of the expression;

name modifier (“@”, “_”, “=” or “?”) determines a type of the expression;

name (consists of arbitrary number of Unicode characters except square brackets “[”, “]” and “.”; first character must not be one of name modifiers characters; can contain parenthesis “(” and “)” but if and only if the opening elements number are equal to closing elements number) is used to identify expression;

index (between square brackets) is an extension of the expression (it can recursively contain another expression);

terminator (dot) is the last character of the expression and is used to concatenate expression with following user-data or expression.

The terminator element can be skipped under special conditions if either:

1. the expression is the last element of the last parameter of the command (no user-data or expression follows it)
 - (a) includes structure where the expression is the last element before block beginning character in block-command;
2. the expression is the last element inside the index of another expression;
3. the expression is the last or unique element of any argument of an operator in complex expressions.

If the user is not certain whether she is able to skip terminator character she safely does not skip it.

³In many known programming languages, the user data is delimited, e.g. using quotes and expressions are not delimited.

8.2 Variables

A *variable* expression is recognized by having name modifier equal to “@” or not having name modifier. T2Script is untyped language with dynamic type checking. Thus, user doesn’t specify variables type explicitly and one variable is able to carry any type of data (value of the variable). The type is checked at run-time when the value of the variable is used with specific commands. During its life-time, one variable can change the type of data it contains.

Variable is either global or local. Global variables don’t have any name modifiers and are visible in all programs scopes. Local variable has name modifier equal to “@” and is visible in a scope of a function it has been set. Local variable persists during function run and then it is automatically removed from the VM’s memory.

Additionally, T2Script supports one-dimensional arrays variables. This type of variables include index after the name. Size of the arrays doesn’t need to be given when the user sets an array and this size may change at runtime. Index must be numeric for some of the commands operate on arrays but T2Script has also limited support (i.e. support for manual operations on them) for associative arrays with indexes that are not numeric.

Different types of variables are presented in listing 8.

Listing 8: Different types of variables in T2Script.

```
// global variable's value
textout $account_number;

// local variable's value
textout @$name;

// global array variable's value
textout $sinus[90];

// global array variable's value with index of local array variable's value
textout $sinus[$@angle[5]];

// associative global array variable's value
textout $birthday[Piotr];
```

8.3 Constants

A *constant* is type of expression that is recognized by having name modifier equal to “_” (underscore). T2Script has build-in set of constants (see table 3). Additionally, application programmer extends this set by defining constants in her application. User is not able to modify constants directly (in opposite to variables) although she can use their current values.

Note, application programmer assigns a string or a function to T2Script constant. Consequently, if function was assigned to a constant, the value of constant may change for subsequent calls. *Constancy* is thus expressed as lack of ability to change the constant value from the script level, not as value’s permanence.

8.4 Complex expressions

A *complex expression* is an expression that has name modifier equal to “?” (question mark) and index which contains *operators* and their *arguments*. Therefore, the main part of complex expression is its index which contains the term to evaluate. The

Constant	Value description
<code>\$_true</code>	true value (zero value)
<code>\$_false</code>	false value (non-zero value)
<code>\$_empty</code>	empty value
<code>\$_parent_name</code>	name of parent command
<code>\$_parent_param</code>	parameters of parent command
<code>\$_owner_name</code>	name of the current command
<code>\$_owner_param</code>	parameters of the current command
<code>\$_time</code>	current time (in local format)
<code>\$_date</code>	current date (in local format)
<code>\$_Pi</code>	Pi value
<code>\$_\n</code> , <code>\$_\r\n</code> and <code>\$_\r</code>	end of line markers
<code>\$_\t</code>	horizontal tab
<code>\$_\\$</code>	dollar sign
<code>\$_\s</code>	space
<code>\$_lparen</code> , <code>\$_rparen</code>	“(“ and “)”
<code>\$_ltabparen</code> , <code>\$_rtabparen</code>	“[“ and “]”
<code>\$_lcurlyparen</code> , <code>\$_rcurlparen</code>	“{“ and “}”
<code>\$_\u(val)</code>	Unicode character of decimal value <i>val</i>

Table 3: Build-in T2Script constants and their values descriptions. In T2Script, false value is represented by 0 (zero character), true value is represented by anything else (including empty string). Parent command is a command that runs current command or is a block-command in which block the current command runs (if command doesn't have a parent, parent-type constants show information of current command).

structure of the index of complex expression is presented in fig. 7.

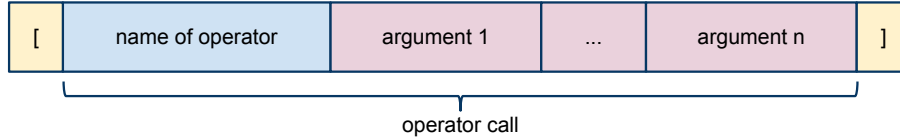


Figure 7: Structure of an index of a generic complex expression. Identical colors of elements express similarity. Each element in operator call is separated by at least one space. Argument of an operator may recursively be another operator call if placed in between parentheses.

Complex expressions call operators and pass them arguments. User uses complex expressions e.g. to evaluate mathematical expressions. The syntax of operators in T2Script is similar to function calls used in functional languages. T2Script however doesn't support lazy evaluation of expressions. Moreover, all the expressions in the complex expression are evaluated before returning the final result. Example of complex expressions usage in T2Script are presented in listing 9.

8.4.1 Operators

Operators are functions (functions are described in section 9) to use with complex expressions. T2Script provides build-in operators that are ready to be used with complex expressions. The operators are divided in 8 categories:

1. Basic arithmetic operators. Include operators for integers and floating point numbers.

Listing 9: Examples of usage of complex expressions.

```
// outputs "101"
textout $?[+ 2 (- 7 8) 100];

// true only if @name value is equal to "Piotr" or "John"
if $?[or (eq $@name Piotr) (eq $@name John)] {
    textout Name correct;
}
```

2. Other arithmetic operators. Include modulo, power, square root, logarithms, minimum, etc.
3. Bitwise operators.
4. Rounding routines operators used for rounding floating point numbers.
5. Logical operators designed for control conditions.
6. Relational operators used to compare strings and numbers and for regular expressions (refer to section 13).
7. Strings operators used to handle strings operations.
8. Other advanced operators. Include variable assignments, existential operator, command execution operator.

The full list of build-in operators with corresponding categories is presented in table 4.

Category	Operators
1.	+, -, *, /, +., -., *., /.,
2.	%, **, sqrt, ln, logn, exp, abs, min, max, tohex
3.	~, &, , ^, <<, >>
4.	round, roundto, ceil, floor
5.	! (not), ?! (or), ?& (and)
6.	== (eq), != (ne), <= (le), >= (ge), < (lt), > (gt), := (eqic), != (neic), =~, =~~, comp
7.	:+ (concat), empty?, len, num?, float?, substr, strpos, strposic, word, char, upcase, downcase
8.	=, exists?, !!, @@, ??

Table 4: List of build-in operators. Operators are separated with coma in the table. If operator has alternative names, they are shown in parentheses.

If user uses operator that is not build-in operator, and if function exists with the name of the used operator, it is called. Thus, user defines own operators functions in the script and calls them with complex expression structure (see section 9.3).

9 Functions

Functions in T2Script are high level objects identified by a name. Function is composed of commands. T2Script shares common structure for definitions of functions and events. The structure is presented in fig. 8.

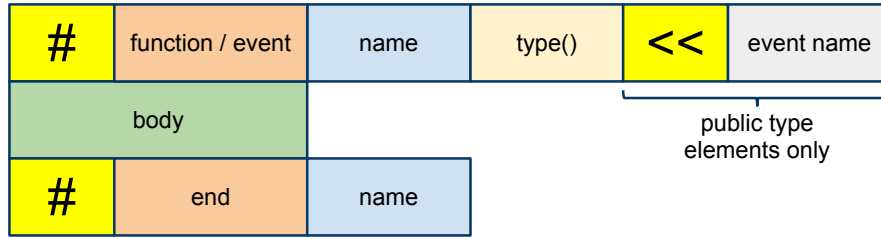


Figure 8: Generic structure for definitions of functions and events. Identical colors of elements express similarity. Some elements (including event name) are only allowed in functions if the type `public()` is chosen.

The function name must be unique for the script file otherwise redefinition error will be signaled to the user. Lines starting with hash must not have semicolon in the end. User chooses the type for the function definition. The type allowed for functions is one of the following types:

private() describes function with no event assigned to it;

public() describes function with one event assigned to it;

operator() is identical to *private()* but describes the function that is used as an operator in complex expressions⁴.

If the function type is *public()*, the existing event name must be provided in order to assign it to the function. In the result, event generator calls the function when the event occurs.

9.1 Functions arguments

Function accepts arbitrary number of arguments. Like in variables, in arguments the type is not specified. When user calls the function, unnamed arguments are available in the function in local array variable `@arg`. The first argument has index equal to 0. When the event generator calls the function, arguments are copied from the event and are usually named accordingly to the content. User may also use `ARGS` command and name unnamed arguments. The `ARGS` command is used only once in the function (usually as the first command). `ARGS` command also enforces minimal number of arguments that must be passed to the function. If the number of arguments passed is smaller than the number expected by `ARGS`, the error is thrown. Example code presenting unnamed and named arguments is presented in listing 10.

Local arguments to the function may also be *accumulated* on the function stack before a call to the function with `PUT` command. This type of accumulated arguments may have arbitrary names. The accumulated arguments are available in the function only in one subsequent call and then are removed with other local variables when function terminates.

9.2 Functions result

Each function returns a result. User alters the default result of function that is equal to `$_empty` (no value) with the following commands:

RETURN sets the result and immediately returns it changing flow of the program;

RESULT sets the result that will be returned when the function terminates without changing flow of the program.

⁴It is not an obligation to use *operator()* type. Essentially, every function can be used as an operator. It is, however, good practice to use it. The script is easier to read.

Listing 10: Unnamed and named arguments in functions.

```
// function uses unnamed arguments
#function unnamed private()
    textout First name: @$arg[0];
    textout Last name: @$arg[1];
#end unnamed

// function uses named arguments
#function named private()
    args first last;
    textout First name: @$first;
    textout Last name: @$last;
#end named

// function uses event's arguments
#function new_user public() << on_new_user
    textout Welcome @$username;
#end new_user
```

9.3 Functions calls

User calls a function to pass the arguments (if any passed) and start execution of function code. There are two modes of function calls:

1. command-mode call when arguments are corresponding to words in a given argument-string (one argument is one space-separated word);
2. call when arguments are not corresponding to words (one argument may be more than one space-separated word).

The first mode is used with command `FUNCTION` and expressions with name modifier equal to “=” and arguments in an index. In this calls, passed argument is in fact one string that is split into words based on spaces – subsequent words are accommodated with subsequent indexes of `@arg` array.

The second type is used with calls of operators – the arguments may contain many words or empty strings. In the example in listing 11, we show the function that returns only first argument passed and is called in different ways with string argument “first second” returning different results based on the call type.

10 Events

User creates own events that she exposes to other scripts and other users. The events in T2Script are based on observer pattern:

- ~> the user creates an event with unique name;
- ~> public function registers itself to be called with an event;
- ~> an event is triggered.

Listing 11: Different types of the functions calls.

```
#function fnc private()
    return $@arg[0];
#end fnc

#function test public() << on_load
    // outputs "first" (type 1)
    textout $=fnc[first second];

    // outputs "first" (type 1)
    function fnc first second;

    // outputs "first second" (type 2)
    textout $?[fnc (concat first $_\s second)];
#end test
```

Events definitions are optional in the language. Application programmer is able to disable them for security reasons. User is not able to trigger build-in events that application programmer defined in the application (using build-in functions of VM).

The event definition is presented in fig. 8. All the naming conditions used for functions (except types) also hold for events. The possible types of events are:

single() when triggered, calls only one, non-deterministically chosen function from the set of registered functions for this event;

multi() when triggered, calls all of the functions from the set of registered functions for this event.

10.1 Events variables

An event structure is very similar to normal function structure. It has local variables. These variables in the moment of triggering are copied to the called registered function(s) with the corresponding names. This mechanism enables user to create events variables. The example of event definition with one event variable *@username* is presented in listing 12.

10.2 Events results

Events return values like other functions. User also checks a result of registered functions by supplying a parameter to TRIGGER command. The parameter becomes a name of local array with values of the results of the registered function(s). In case the event has *single()* type, the array contains only one element. In case there are no registered functions for the event, the array is not set. Example of event giving an approval for an action is presented in listing 13.

The presented example first sets the result to *\$_true*, then triggers an event and calls all registered functions. The array *@votes* is used to store the results from the functions. If any of the registered function returns *\$_false*, then the event does not give an approval for an action (returns *\$_false*). Function *block_shutdown* is an example of registered function. It checks two actions: “shutdown” and “exit” with WHITELIST command and if program is processing while one of these actions is triggered, it returns *\$_false*.

Listing 12: Example of event definition.

```
#event on_new_user multi()  
    args username;  
    trigger;  
#end on_new_user  
  
#function create_user private()  
    // here creation of new user "Piotr"  
    exp $=on_new_user[Piotr];  
#end create_user
```

11 Timers

Timers are parts of program that repeat the execution automatically based on given time interval. Timer is recognized by its name. If user sets a name of the timer to “auto”, the VM generates timer name automatically. Timers operations are handled with few timers-related commands including block command SETTIMER. SETTIMER creates and runs the timer immediately. The example of timer is presented in the listing 14.

Timer copies local variables (from function or event) to its local scope in the moment of creation. Therefore, in the example, a *@local* variable value change after creation of the timer (automatic name, 1000 millisecond interval and 10 iterations) is not affecting the value of *@local* in the timer. The mechanism of copying the variables is similar to copying mechanism used in events when calling registered functions (see section 10.1).

Part III

Advanced programming techniques

12 Exceptions handling

An *exception* in T2Script is an error message in the script. When the exception propagates, a current program execution is stopped and the exception is presented for the user. User may ignore the exception by catching it. User catches the exception with command CATCH. If the parameter is present in CATCH command, user saves the exception to variable for error examination purposes from the script level. Additionally, user may throw own exception from the variable using THROW command. The example program using exceptions is presented in listing 15.

The presented example first throws an error and then catches it and display it to the user an error message. The error propagation mechanism presented in the example is intra-functional but it is valid also for inter-functional scenarios – the errors propagate between the functions calls.

Listing 13: Example of an event returning a result.

```
#event on_approval multi()
    args action;
    // default result
    result $_true;
    trigger @votes;
    foreach @vote in @votes {
        if $?[^ $@vote] {
            return $_false;
        }
    }
#end on_approval

#function block_shutdown public() << on_approval
    whitelist @action shutdown exit;
    if $processing {
        return $_false;
    }
#end block_shutdown
```

13 Regular expressions

T2Script includes extended support for regular expressions with commands and operators. Regular expressions in T2Script are based on *Boost Regex Library* [6]. The syntax of the regular expressions is Perl syntax and follows *Regex* manual⁵. The regular expressions syntax underlies the T2Script syntax – if a regular expressions uses reserved T2Script literals, these must be escaped prior to passing them to regular expression.

14 Metaprogramming

T2Script VM supports metaprogramming techniques with command `ENVRS`. The command execution includes:

- ~> execute any other interpreter with given parameters,
- ~> pipeline the result back to VM,
- ~> evaluate the result as T2Script.

The process of calling and pipelining is optimized and the additional cost caused by using an external interpreter is minimized. T2Script programs that are the result of this process do not pass through the whole process of compilation but through the process called *minimal-compilation*. Noticeably, scripts used with minimal-compilation must not contain definitions of events or functions and are executed with a default context (see section 7.4). The process is illustrated in the fig. 9.

⁵http://www.boost.org/doc/libs/1_45_0/libs/regex/doc/html/boost_regex/syntax/perl_syntax.html

Listing 14: Timer example.

```
#function counter private()
    setvar @local Local variable;
    settimer auto 1000 10 {
        textout @$local;
    }
    setvar @local Hello;
#end counter
```

Listing 15: Exceptions handling.

```
#function fnc private()
    catch @err {
        setvar @msg This is error message;
        throw @msg;
    }
    textout @$err;
#end fnc
```

15 Dynamic-Link Libraries (DLL) plugins

T2Script has internal support for DLL libraries including loading and unloading DLL files as well as mapping DLL functions into T2Script functions. The mapping converts the local function variables to C++ array and maps back the result of C++ function into T2Script function's result using one command `DLLLOCAL`. This mechanism allows user to write plugins to applications that include T2Script module.

Additionally, the application programmer may allow user to write callback plugins using one of the mechanisms that is provided by specific operation system where the application is deployed (e.g. message queries, shared memory).

16 Debugging and Testing

T2Script VM offers API for attaching the debugger at *command level*. Debugger provides command by command execution without partial expression execution. The debugger offers also watch-lists of all functions, global variables and timers. Local variables are evaluated during the debugging process inside a function. The T2Script module does not offer any GUI for debugging, it must be provided by application programmer.

Language does not offer any test suite but it can be integrated without afford by application programmer.

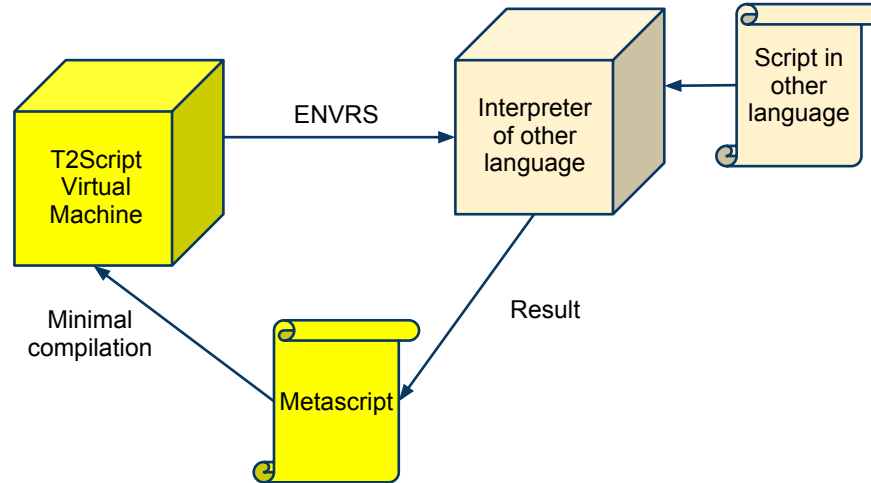


Figure 9: Metaprogramming technique with T2Script. The metascript does not compile in normal compilation process but instead passes minimal-compilation that minimizes the cost of using an external interpreter of other language by reducing compilation time.

Part IV

Summary

17 Solutions to the given problems

After the description of the language, we show the solution to the problems given in section 2.

17.1 Interaction with the user

In CED programming the user interacts with the language by commands and scripts with events definitions and functions. Therefore, the language increases the interaction with the users. It is reflected in T2Script language construct of command that provides natural way of expressing the name and parameters.

17.2 Abstraction of an events dispatcher

We would like our language to abstract the events and dispatcher on the language level. Therefore the design of the language includes the events definition and events-registered functions (see section 10). The events are implicit parts of the language and user is encouraged to use them even for simple programs.

17.3 Separation of the text and expressions

T2Script is mainly designed to operate on string data. As it was shown in section 8.1 and description of expressions format, the expressions are separated from the text, not the text from expressions (difference of importance). This is an experimental design that we chose for our string handling language.

17.4 Other languages modules

We would like our language to be capable to easy extend with modules written in other programming languages. Moreover, we would like the language to do this with optimal cost. The solution for this was presented in sections 14 and 15. Built-in T2Scripts commands allow user to apply metaprogramming techniques using other languages interpreters and to write simple DLL plugins. The minimal-compilation process optimizes the cost of metaprogramming scripts run times.

17.5 Static code analysis

By usage of CED Programming code security issue is solved without static analysis of the application. The solution we propose promotes enforcement of contract during the run-time in the VM.

The description of disabling the selected commands of T2Script language is presented in section 7.7. The application is able to apply the contract at run-time by disabling specific VM commands prior to unknown script execution. That reduces the risk of executing a potentially dangerous script. Contract schema is however not included in T2Script so this problem is not fully solved.

17.6 Language extensions

By the same mechanisms that are used to disable selected commands, the new commands are added by application programmer. This enables the language extensibility with possibilities of fast creation of new language structures etc. Furthermore, the user also extends the language in limited ways by defining own operators (see section 8.4.1).

18 CED programming

Summarizing, we define CED programming as a paradigm in which programming language:

1. gives user natural way of using commands that are part of the language;
2. incorporates events handling mechanisms.

In CED programming, everything is based on command entity and events are used to invoke groups of commands.

19 Evaluation

T2Script is dynamic untyped language (or scripting language). The comparison of system and scripting programming languages reveals that scripting languages shorten the development time from 4 to 8 times in average and reduce number of lines from 100 to 1000 times in comparison to system programming languages [9]. Thus T2Script effectively reduces the workload needed to write scripts based on commands and expressions.

T2Script module is designed to be embedded into custom application in 24 hours work period of experienced programmer. It is noticeably short time in comparison to the time needed to implement own scripting module. Unicode format of characters used in T2Script allows internationalization.

T2Script offers natural approach in handling application commands, network and interfaces. Commands nowadays are used in some form in almost all applications. The nature of commands however is very simple and may an obstacle when implementing complex programs – this is why T2Script introduces block-commands available from script files. The T2Script command parameters (except the last one) correspond to words in the user-string. This simple approach might be an advantage or a disadvantage dependent on the application.

T2Script commands are relatively easy to understand in comparison to expressions that require more attention from a user. User has wide possibilities of writing modules containing events and later to expose them to other users. It encourages the code reusability and modularity.

T2Script offers many extensions possibilities using metaprogramming techniques or DLL plugins and therefore gives the user a powerful tool while relieving application programmer.

20 Future work

In this document, we mentioned few times how application programmer integrates T2Script module into custom application. This topic however should be extended in the next publications to be discussed in more details.

We can see the possibility of future work in investigation of the syntax and performance measures of T2Script in comparison to other programming languages.

Another possible direction is implementing a test suite for the users in T2Script module. The test suite can adopt existing commands and VM functions used now for debugging. It will however require tests at different levels and include an event simulator.

As the T2Script files are compiled to a bytecode, this bytecode format can be written to a file. Currently the compilation occurs internally but, in the future, it may be possible to give the user an opportunity to save the scripts files in bytecode format.

Another possible future extension is to build a standardize processes of contract enforcement and assignment for T2Script, so that the user will trust the unknown scripts during the execution.

References

- [1] J. Aliprand, J. Allen, J. Becker, M. Davis, M. Everson, A. Freytag, J. Jenkins, M. Ksar, R. McGowan, E. Muller, et al. The Unicode standard, version 4. 0. *Computing Reviews*, 45(8):480, 2004.
- [2] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.
- [3] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.
- [4] S. Ferg. Event-driven programming: Introduction, tutorial, history, 2006.
- [5] A. Ghosal, T.A. Henzinger, C.M. Kirsch, and M.A.A. Sanvido. Event-driven programming with logical execution times. *Hybrid Systems: Computation and Control*, pages 167–170, 2004.
- [6] Björn Karlsson. *Beyond the C++ Standard Library*. Addison-Wesley Professional, 2005.
- [7] J.C. Mitchell. *Concepts in programming languages*. Cambridge University Press, 2003.
- [8] J.K. Ousterhout. *Tcl: An embeddable command language*. University of California at Berkeley, 1989.
- [9] J.K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 2002.
- [10] B.R. Rau. Levels of representation of programs and the architecture of universal host machines. In *Proceedings of the 11th annual workshop on Microprogramming*, pages 67–79. IEEE Press, 1978.
- [11] J. Sametinger. *Software engineering with reusable components*. Springer Verlag, 1997.
- [12] J.E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.